Java Bytecode Specification and Verification

Lilian Burdy
INRIA Sophia-Antipolis
2004 route des Lucioles BP 93
06902 Sophia-Antipolis, France
Lilian.Burdy@sophia.inria.fr

Mariela Pavlova
INRIA Sophia-Antipolis
2004 route des Lucioles BP 93
06902 Sophia-Antipolis, France
Mariela.Pavlova@sophia.inria.fr

ABSTRACT

We propose a framework for establishing the correctness of untrusted Java bytecode components w.r.t. to complex functional and/or security policies. To this end, we define a bytecode specification language (BCSL) and a weakest precondition calculus for sequential Java bytecode. BCSL and the calculus are expressive enough for verifying non-trivial properties of programs, and cover most of sequential Java bytecode, including exceptions, subroutines, references, object creation and method calls.

Our approach does not require that bytecode components are provided with their source code. Nevertheless, we provide a means to bring the benefits of source program specifications to the code consumer by defining a compiler from the Java Modeling Language (JML) to BCSL. Our compiler can be used in combination with many Java compilers to produce extended class files from JML-annotated Java source programs.

All components, including the verification condition generator and the compiler are implemented and integrated in the Java Applet Correctness Kit (JACK), and have been tested on non-trivial examples.

1. INTRODUCTION

This paper addresses the problem of establishing trust in untrusted software components. This question concerns important areas like smart card applications, mobile phones, bank cards, ID cards and whatever scenario where untrusted code should be installed and executed. In particular, depending on what is the level of trust the code receiver wants to establish, the state of the art proposes different solutions. For example, the verification may be performed over the source code. In this case, the code receiver should make the compromise to trust the compiler, which is problematic. The bytecode verification technique proposes another solution, which requires no tradeoff with the compiler. The bytecode verifier performs the static analysis directly over the bytecode and guarantees that the code is well typed and well formed. Yet, bytecode verification is limited to properties, that only guarantee that the bytecode does not violate the proper function of the virtual machine. Another solution is the Proof Carrying Code paradigm (PCC) and the certifying compiler. In this architecture, the untrusted code is accompanied by a proof for its safety w.r.t. to some safety property and the code receiver has just to generate the verification conditions and type check the proof against them. The proof is generated automatically by the certifying compiler and thus for properties like well typedness or safe memory access. As the certifying compiler is designed to be completely automatic, it will not be able to deal with rich functional or security properties.

The present work proposes an interactive verification framework for establishing trust between a client and a code producer against non trivial security or functional policies, where the untrusted code is likely to lack its source code. More generally, the frameweork can be applied to scenarios where the code producer has to implement or fulfill some client requirements. Using this framework, the producer can generate and supply, along the bytecode, the specification information sufficient for the client to establish that the code respects those requirements.

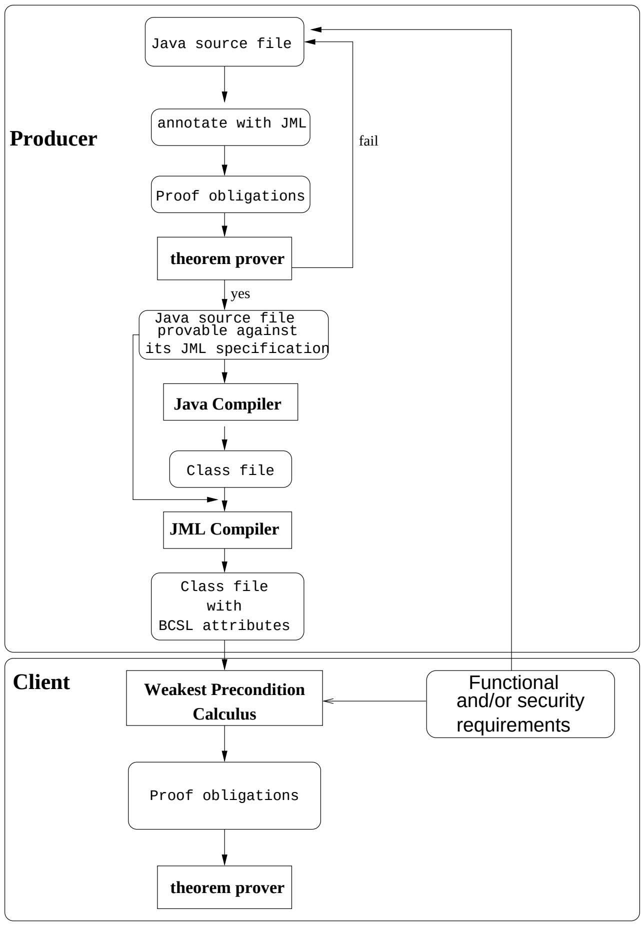
In particular, the architecture is tailored to Java bytecode. The Java technology finds a large application in mobile and embedded components because of its portability across platforms. For instance, its dialect JavaCard is largely used in smartcard applications and the J2ME Mobile Information Device Profile (MIDP) finds application in GSM mobile components. In this article we propose a static verification technique using formal methods for sequential Java bytecode programs.

The aforementioned scheme is composed by several coponents. We define a bytecode logic in terms of weakest precondition calculus for the sequential Java bytecode language. The logic rules out almost all Java bytecode instructions and supports the Java specific features like exceptions, references, method calls and subroutines. We define a bytecode specification language, called BCSL, and supply a compiler from the high level Java specification language JML [13] to BCSL. BCSL supports a JML subset which is expressive enough to specify rich functional properties. The specification is inserted in the class file format in newly defined attributes, thus making not only the code mobile but also its specification. These class file extensions do not affect the JVM performance. The scheme makes the Java bytecode benefit from the specification written at source level. We have implementations of a verification condition generator based on the weakest precondition calculus and of the JML specification compiler. Both are integrated in the Java Applet Correctness Kit (JACK) [9].

In what follows, the sections and their corresponding topics are: Section 2 reviews scenarios in which the architecture is appropriate to use; Section 3 provides an overview of related work; Section 4 gives background information about the JML specification language; Section 5 presents the bytecode specification language BCSL and the JML compiler; Section 6 explains how the weakest precondition calculus works illustrating it with definitions and example; in this section we also give the verification conditions that are generated for proving program correctness; Section 7 concludes with future work.

2. FRAMEWORK

Figure 1 presents the proposed overall architecture for ensuring Java bytecode correctness. It describes a process that



Producer
Java source file
annotate with JML
fail
Proof obligations
theorem prover
yes
Java source file provable against its JML specification
Java Compiler
Class file
JML Compiler
Class file with BCSL attributes
Client
Weakest Precondition Calculus
Functional and/or security requirements
Proof obligations
theorem prover

Figure 1: The overall architecture for annotating and verifying code

allows a client to trust a code produced by an untrusted code producer.

In the first stage of the process the client provides the functional and (or) security requirements to the producer. The requirements can be in different form:

- A specified interface that describes the application to be developed. In that case, the client has fully specified in JML the features that have to be implemented by the code producer.
- An API with some restricted access to some method. In this case, the client can protect its system by restricting its usage (for example, if the client API provides transaction management facilities, a requirement can be for no nested transactions and the API method `open` for opening and method `close` for closing transactions can be annotated to ensure that `close` should not be called if there is no transaction running and `open` should not be called if there is already a running transaction).

In the development process, the producer verifies if the client requirements are respected by generating verification conditions over the source code and usually, he has to add JML annotations for this e.g. loop invariants, class invariants, method preconditions and postconditions etc. Usually, only after specifying enough the source code, have we got the annotated Java source files to feed to the JML compiler.

When the annotations are sufficient to prove the code, the Java file is then normally compiled with a Java compiler to obtain a class file. This class file is then extended with user defined attributes that contain the BCSL specification, resulting from the compilation of the JML specification. At this stage, the Java class files contain all the information that will allow the client to check it. In particular, the client will generate proof obligations from the untrusted annotated bytecode and his security requirements (expressed in a suitable form) as shown in figure 1. Proof obligations are formulas which, if provable, guarantee the bytecode correctness. The latter are then proved with a theorem prover (possibly interactively). If the client succeeds in proving the verification conditions, he can trust the unknown code. Currently the framework does not support sending both the proof and the bytecode to the client, which is the next step in our work.

To implement this architecture we use JACK as a verification condition generator both on the consumer and the producer side. JACK is a plugin for the eclipse[1] integrated development environment for Java. Originally, the tool was designed as verification condition generator for Java source programs against their JML specification. JACK can interface with several theorem provers (AtelierB, Simplify, Coq, PVS). We have extended the tool with a compiler from JML to BCSL and a bytecode verification condition generator. In the following we introduce the BCSL language, the JML compiler and the bytecode weakest precondition calculus which underlines the bytecode verification condition generator.

[1]http://www.eclipse.org

3. RELATED WORK

There are several fields which are related to the present work: bytecode verification, logic for unstructured program languages, attaching specification to the compiled code, architecture for checking untrusted components.

Bytecode verification is concerned with establishing that a bytecode is well typed (every instruction is applied to operands of the correct type) and well formed (e.g. no jumps to an un-existing bytecode index), differently from the goals of the present work where program correctness is defined in terms of functional correctness. The Java Virtual Machine (JVM), for example, is provided with a bytecode verifier. The field is well researched and for more information one can look at [14].

Floyd is among the first to work on program verification for unstructured languages (see [22]). Few works have been dedicated to the definition of a bytecode logic. In [20], Quigley defines a Hoare logics for bytecode programs. This work is limited to a subset of the Java virtual machine instructions and does not treat for example method calls, neither exceptional termination. The logic is defined by searching a structure in the bytecode control flow graph, which gives an issue to complex rules.

A work close to ours is presented in [3] by P.Muller and F.Bannwart. The authors define a Hoare logic over a bytecode language with objects and exceptions. A compiler from source proofs into bytecode proofs is also defined. As in our work, they assume that the bytecode has passed the bytecode verification certification. The bytecode logic aims to express functional properties. To our knowledge subroutines are not treated. Invariants are inferred by fixpoint calculation, differently from the approach presented here, where invariants are compiled from the high level JML specification (see section 5.2). However, infering invariants is not a decidable problem. The work of P.Muller and F.Bannwart is inspired by the Nick Benton's work (see [5]). In the latter a bytecode logic for a stack based language is defined which checks programs both for well — typedness and functional correctness. The language does not support objects, references, exceptions neither subroutines.

In [23], M. Wildmoser and T. Nipkow describe a framework for verifying Jinja (a Java subset) bytecode against arithmetic overflow. The annotation is written manually, which is not comfortable, especially on bytecode. Here we propose a way to compile a specification written in a high level language, allowing specification to be written at source level, which we consider as more convenient.

The Spec# ([4]) programming system developed at Microsoft proposes a static verification framework where the method and class contracts (pre, post conditions, exceptional postconditions, class invariants) are inserted in the intermediate code . Spec# is a superset of the C# programming language, with a built-in specification language, which proposes a verification framework (there is a choice to perform the checks either at runtime or statically). The static verification procedure involves translation of the contract specification into metadata which is attached to the intermediate code and the verification is performed over the bytecode by the Boogie theorem prover.

Another topic related to the present work is PCC. PCC and the certifying compiler were proposed by Necula (see [16, 17, 18]). PCC is an architecture for establishing trust in untrusted code in which the code producer supplies a proof for correctness with the code. The initial idea for PCC was that the producer automatically infers annotation for properties like well typedness, correct read/writes and automatically generates the proof for their correctness using the certifying compiler. Such properties guarantee that a program do the things correctly and not that it does the right things. The present work is targeting at a framework for establishing complex functional and interface properties whose automatic checking is hard and even impossible. From the above cited papers, [3] is aiming also at building PCC for guaranteeing not trivial properties. As we stated in section 1, our framework currently does not support adding proofs to bytecode which but we consider this point as a future work.

4. A QUICK OVERVIEW OF JML

JML [13] (short for Java Modeling Language) is a behavioral interface specification language tailored to Java applications. JML follows the design-by-contract approach (see [6]), where classes are annotated with class invariants and method pre- and postconditions. Specification inside methods is also possible; for example one can specify loop invariants, or assertions predicates that must hold at specific program points.

JML specifications are written as comments so they are not visible by Java compilers. The JML syntax is close to the Java syntax: JML extends Java with few keywords and operators. For introducing method precondition and postcondition one has to use the keywords `requires` and `ensures` respectively, `modifies` keyword is followed by all the locations that can be modified by the method, `loop_invariant`, not surprisingly, stands for loop invariants, `loop_modifies` keyword gives the locations modified by loop invariants etc. The latter is not standard in JML and is an extension introduced in [9]. Special JML operators are, for instance, `\result` which stands for the value that a method returns if it is not void, the `\old(expression)` operator designates the value of `expression` in the prestate of a method and is usually used in the method's postcondition. JML also allows the declaration of special JML variables, that are used only for specification purposes. These variables are declared in comments with the `model` modificator and may be used only in specification clauses.

JML can be used for either static checking of Java programs by tools such as JACK, the Loop tool, ESC/Java [1] or dynamic checking by tools such as the assertion checker jmlrac [12]. An overview of the JML tools can be found in [8].

Figure 2 gives an example of a Java class that models a list stored in a private array field. The method `replace` will search in the array for the first occurence of the object `obj1` passed as first argument and if found, it will be replaced with the object passed as second argument `obj2` and the method will return true; otherwise it returns false. The loop in the method body has an invariant which states that all the elements of the list that are inspected up to now are different from the parameter object obj1. The loop specification also states that the local variable i and any element of the array field list may be modified in the loop.

```
public class ListArray {
  private Object[] list;

  //@requires list != null;
  //@ensures \result ==(\exists int i;
  //@ 0 <= i && i < list.length &&
  //@ \old(list[i]) == obj1 && list[i] == obj2);
  public boolean replace(Object obj1,Object obj2)
  {
    int i = 0;
    //@loop_modifies i, list[*];
    //@loop_invariant i <= list.length && i >=0
    //@  && (\forall int k;0 <= k && k < i ==>
    //@  list[k] != obj1);
    for (i = 0; i < list.length; i++ ) {
      if ( list[i] == obj1) {
        list[i] = obj2;
        return true;
      }
    }
    return false;
  }
}
```

Figure 2: class ListArray with JML annotations

5. BYTECODE SPECIFICATION LANGUAGE (BCSL)

In this section, we propose a bytecode specification language which we call BCSL. We define a compiler from the high level specification language JML to BCSL. The specification compilation results in a class file extension. In the following we give the grammar of BCSL and sketch the specification compiler.

5.1 Grammar

We propose a bytecode level specification language which corresponds to a representative subset of JML. We sketch the bytecode specification language grammar in figure 3. We omit some of the definitions because of space constraints, e.g. the grammar for arithmetic expressions (which is defined in a standard way). The full specification can be found in [10]. The language defined here is expressive enough for most purposes including the description of non trivial functional and security properties. We now discuss some of the specification clauses that have some differences with JML, for the rest their semantics is the same as in JML and can be found in [21, 13].

We can specify using the specification clause exsures what is the postcondition of a method in case it terminates with an exception E. If the postcondition states something about the exception object thrown then the special expression EXC is used (this expression can appear only in exceptional postconditions). If exsures is not specified for certain exception, then by default it is considered as *false*.

ClassSpec ::= class invariant $\mathcal{P}$
| history constraint $\mathcal{P}$
| model ClassName id

MethodSpec ::= SpecCase
| SpecCase also MethodSpec;

SpecCase ::= requires $\mathcal{P}$;
modifies $list(\mathcal{E})$;
ensures $\mathcal{P}$;
exsures (ExceptionClass) $\mathcal{P}$;

InterMethodSpec ::= loopSpec
| assertSpec

loopSpec ::= pc_ index int;
loop_modifies $list(\mathcal{E})$;
loop_invariant $\mathcal{P}$;
loop_decreases $\mathcal{E}$;

assertSpec ::= pc_ index int;
assert $\mathcal{P}$;

$\mathcal{P}$ ::= true | false
| $\mathcal{E}$ $predSymbol$ $\mathcal{E}$
| $\mathcal{P} \wedge \mathcal{P}$ | $\mathcal{P} \vee \mathcal{P}$ | $\mathcal{P} \Rightarrow \mathcal{P}$
| $\forall$(boundVar : $JavaType$)$\mathcal{P}$
| $\exists$(boundVar : $JavaType$)$\mathcal{P}$

$\mathcal{E}$ ::= Arithmetic_Expr | lv[i]
| ref | int_literal
| field_cp_index($\mathcal{E}$) | $\mathcal{E}[\mathcal{E}]$
| \result | \old($\mathcal{E}$) | EXC
| \typeof($\mathcal{E}$) | null | this
| c | st(Arithmetic_Expr) ...

Figure 3: BCSL grammar

As shown by the grammar, BCSL allows to specify different method specification cases separated by the keyword also — this means that method caller has to satisfy the disjunction of the preconditions in the specification cases and the method's implementation has to guarantee the postcondition of every specification case of which the precondition held in the prestate.

Loop specifications and assertions are tagged with the program point in the bytecode where they must hold. Among the expressions that are handled (almost all are also handled by JML) we have the expressions : lv[i] standing for the i-th local variable in a method; field_cp_index($\mathcal{E}$) meaning a field access expression, where field_cp_index is the constant pool index describing the field; c stands for the stack counter and st(Arithmetic_Expr) a stack element at position Arithmetic_Expr. These expressions do not appear in the precondition and postcondition specification of a method. Later we shall see how they are used.

5.2 Compiling JML into bytecode specification language

This section explains how JML specifications are compiled into bytecode level specifications and how they are inserted into the bytecode.

Before going farther we give a brief description of the class file format. As defined by the Java Virtual Machine Specification (JVMS) [15], a class file contains a definition of a single class class or interface. It contains information about the class name, interfaces implemented by the class, super class, methods and fields declared in the class and references. The JVMS mandates that the class file contains data structure usually referred as the **constant_pool** table which is used to construct the runtime constant pool upon class or interface creation. The runtime constant pool serves for loading, linking and resolution of references used in the class. The JVMS allows to add to the class file user specific information([15], ch.4.7.1). This is done by defining user specific attributes (their structure is predefined by JVMS).

Thus the "JML compiler" [2] compiles the JML source specification into user defined attributes. The compilation process has three stages:

1. compile the Java source file. This can be done by any Java compiler that supplies for every method in the generated class file the **Line_Number_Table** and **Local_Variable_Table** attributes. The presence in the Java class file format of these attribute is optional [15], yet almost all standard non optimizing compilers can generate these data. The **Line_Number_Table** describes the link between the source line and the bytecode of a method. The **Local_Variable_Table** describes the local variables that appear in a method. Those attributes are important for the next phase of the JML compilation.

2. from the source file and the resulting class file compile the JML specification. In this phase, Java and JML source identifiers are linked with their identifiers on bytecode level, namely with the corresponding indexes either from the constant pool or the array of local variables described in the **Local_Variable_Table** attribute. If in the JML specification a field identifier appears, for which no constant pool (cp) index exists, such is added in the constant pool and the identifier in question is compiled to the new cp index. It is also in this phase that the specification parts like the loop invariants and the assertions which should hold at a certain source program point must be associated to the respective program point on bytecode level. The specification is compiled in binary form using tags in the standard way. Basically the compilation of an expression is a tag followed by the compilation of its subexpressions. Thus for example the loop invariant specified in JML for the method `replace` in figure 2 is :

$$\begin{array}{l}\texttt{lv[3]} \leq \mathit{length}(\#19(\texttt{lv[0]})) \wedge \\ \texttt{lv[3]} \geq 0 \wedge \\ \forall v_0 \in \mathit{int}. \left(\begin{array}{l} 0 \leq v_0 \wedge \\ v_0 < \texttt{lv[3]} \\ \Rightarrow \#19(\texttt{lv[0]})[v_0] \neq \texttt{lv[1]} \end{array}\right)\end{array}$$

From the example one can see that local variables and fields are respectively linked to the index of the register table for the method and to the corresponding index of the constant pool table (#19 is the compilation of the field name `list`, `lv[3]` stands for the method local variable `i`).

3. add the result of the JML compilation in the class file as user defined attributes. Method specifications, class invariants, loop invariants are newly defined attributes in the class file. For example, the specification of all the loops in a method are compiled to a unique method attribute: whose syntax is given in figure 4. This attribute is an array of data structures each describing a single loop from the method source code. Also for each loop in the source code there must be a corresponding element in the array. More precisely, every element contains information about the instruction where the loop starts as specified in the **Line_Number_Table**, the invariant associated to this loop, the decreasing expression in case of total correctness, the expressions that can be modified. For the full specification of the compiler see [10].

```
JMLLoop_specification_attribute {
    ...
    {  u2 index;
       u2 modifies_count;
       formula modifies[modifies_count];
       formula invariant;
       expression decreases;
    } loop[loop_count];
}
```

- **index**: The index in the `LineNumberTable` where the beginning of the corresponding loop is described
- **modifies**[]: The array of modified expressions.
- **invariant** : The predicate that is the loop invariant. It is a compilation of the JML formula in the low level specification language
- **decreases**: The expression whose decreasing at every loop iteration

Figure 4: Structure of the Loop Attribute

The JML compiler does not depend on any specific Java compiler, but it requires the presence of a debug information, namely the presence of the **Line_Number_Table** attribute for the proper compilation of inter method specification, i.e. loops and assertions. We think that this is an acceptable restriction for the compiler. The most problematic part of the compilation is to find the program points

[2]Gary Leavens also calls his tool jmlc JML compiler, which transforms jml into runtime checks and thus generates input for the jmlrac tool

where the loop invariants must hold. This basically means that one has to identify which source loop corresponds to which bytecode loop in the control flow graph. To do this, we assume that the control flow graph is reducible (see [2]); intuitively this means no jumps from outside a loop inside it; graph reducibility allows to establish the same order between loops in the bytecode and source code level and to compile correctly the invariants to the proper places in the bytecode.

limitations : registers that are used with two different types in the method bytecode

6. WEAKEST PRECONDITION CALCULUS FOR JAVA BYTECODE

In this section, we define a bytecode logic in terms of a weakest precondition calculus. We assume that the bytecode program has passed the bytecode verification procedure (we discuss the issue in section 3), thus the calculus is concerned only with program functional properties. We also assume that code is generated by a non optimizing compiler.

The proposed weakest precondition has those features:

- it supports all Java sequential instructions except for floating point arithmetic instructions and 64 bit data (`long` and `double` types), including exceptions, object creation, references and subroutines. The calculus is defined over the method control flow graph
- it supports BCSL (section 5), i.e. method's specification written in BCSL like pre- and postconditions, assertions at particular program point among which loop invariants (if there is nothing special specified the specification by default: preconditions, postconditions and invariants are taken to be true) is taken into account.

The calculus is defined over the control flow graph of the program and has two levels of definitions — the first one is the set of rules for single Java bytecode instructions (discussed in subsection 6.1) and the second one takes into account how control flows in the bytecode (subsection 6.3). A related problem is how the loops in the control flow are treated. As we mentioned earlier we assume that every method is specified in sufficient details, i.e. for each loop, the corresponding loop invariant is present. This allows us to "cut" the loops in the graph at the program point where the invariant must hold. These "cuts" generate an abstract control flow graph which is acyclic and over which the verification conditions are generated. Subsection 6.2 discusses how the abstract control flow graph is generated.

6.1 Weakest Precondition for bytecode instructions

We define a weakest precondition (wp) predicate transformer function which takes into account normal and exceptional termination. wp takes three arguments: an instruction, a predicate that is the instruction's normal postcondition ψ, and a function from exception types to predicates ψ^{exc} (it returns the specified postcondition in the `exsures` clause for a given exception; see section 3). The wp function returns the weakest predicate such that if it holds in the state when the instruction starts its execution the following conditions are met:

1. if the instruction terminates its execution normally the predicate ψ holds in the poststate
2. if it terminates with an exception `E` then the predicate ψ^{exc}(`E`) holds in the poststate.

So the signature of wp is:

$$wp : \texttt{instruction} \rightarrow \mathcal{P} \rightarrow (\texttt{ExceptionType} \rightarrow \mathcal{P}) \rightarrow \mathcal{P}$$

The Java bytecode language is stack based, i.e. the instructions take their arguments from the method execution stack and put the result on the stack. So, we have to represent the elements of the stack that correspond to the arguments and the result for a given instruction. In the examples in figure 5 `st(c)` stands for the element on the top of the stack. The instruction `Type_load` i loads on the top of the stack the value of the method local variable at index i in the **Local_Variable_Table** (see section 5). The wp rule for `Type_load i` increments the stack counter `c` and puts on `st(c)` the contents of the local variable `lv[i]`. As we said in the beginning of the section, wp "understands" the bytecode specification language, i.e. the keywords have their corresponding semantics. For example, the keyword `\result` is evaluated only by `Type_return` instructions and if appearing in the postcondition, `\result` is substituted by the element on the top of the stack `st(c)`.

The rules also take into account the possible abnormal execution of the instruction. For example, in Fig. 5, the rule for the instruction `putField` has two conjuncts - one in the case when the dereferenced object is not null and the instruction execution terminates normally; the other one stands for the case when this is not true. Note, that if the exception thrown is not handled, we substitute the special specification variable `EXC` (see Subsection 5.1) in the exceptional postcondition by the thrown exception object.

A complete definition of wpcan be found in [11].

6.1.1 *References: manipulating fields*

Instance fields are treated as functions, where the domain of a field `f` declared in the class `Cl` is the set of objects of class `Cl` and its subclasses. We are using function update when assigning a value to a field reference as, for instance in [7]. In Fig. 5 the rule for `putField` substitutes the corresponding field function `Cl.f` with `Cl.f` updated for object o, in case the dereferenced object is not null. The definition of update function is given in figure 6.

6.1.2 *Method calls*

Method calls are handled by using their specification. A method specification is a contract - the precondition of the called method must be established by the caller at the program point where the method is invoked and its postcondition is assumed to hold after the invocation. The rule for

$$wp(\texttt{iinc}\ i,\ \psi,\ \psi^{exc}) = \psi[\texttt{lv[i]} \leftarrow \texttt{lv[i]} + 1]$$

$$\begin{array}{l} wp(\texttt{Type\_load}\ i,\ \psi,\ \psi^{exc}) = \\ \quad \psi[\texttt{c} \leftarrow \texttt{c}+1][\texttt{st(c+1)} \leftarrow \texttt{lv[i]}] \\ \quad \texttt{where } i \texttt{ is a valid local variable index} \end{array}$$

$$\begin{array}{l} wp(\texttt{putField Cl.f},\ \psi,\ \psi^{exc}) = \\ \quad \texttt{st(c-1)} \neq \texttt{null} \\ \quad \Rightarrow \quad \psi[\texttt{c} \leftarrow \texttt{c} - 2] \\ \qquad [\texttt{Cl.f} \leftarrow \texttt{Cl.f} \oplus [\texttt{st(c-1)} \rightarrow \texttt{st(c)}]] \\ \quad \wedge \\ \quad \texttt{st(c-1)} = \texttt{null} \\ \quad \Rightarrow \phi[\texttt{c} \leftarrow 0][\texttt{st(0)} \leftarrow \texttt{st(c)}] \end{array}$$

where the predicate ϕ is the precondition of the exception handler protecting the instruction against `NullPointerException` *if it exists, otherwise if the* `NullPointerException` *is not handled*
$\phi = \psi^{exc}(\texttt{NullPointerException})[\texttt{EXC} \leftarrow \texttt{st(c)}]$

Figure 5: rules for some bytecode instructions

$$(\texttt{Cl.f}) \oplus [e2 \rightarrow e1](o) = \begin{cases} e1 & \textit{if } e2 = o \\ \texttt{Cl.f}(o) & \textit{else} \end{cases}$$

Figure 6: Overriding Function

$$\begin{array}{l} wp(\texttt{invoke}\ \ \texttt{m},\ \psi,\ \psi^{exc}) = \\ \quad \psi^{pre}(m) \wedge \\ \quad \forall_{j=1..s} e_j.(\\ \quad \psi^{post}(\texttt{m})[\texttt{lv[i]} \leftarrow \texttt{st(c + i - numArgs(m))}]_{i=0}^{\texttt{numArgs(m)}} \\ \qquad\qquad [\texttt{\textbackslash result} \leftarrow \texttt{fresh\_var}] \\ \quad \Rightarrow \psi[\texttt{c} \leftarrow \texttt{c} - \texttt{numArgs(m)}][\texttt{st(c)} \leftarrow \texttt{fresh\_var}]) \wedge_{\texttt{i}=1}^{\texttt{k}} \\ \quad \forall_{j=1..s} e_i(\\ \quad \psi^{exc}_{\texttt{m}}(\texttt{Exc}_\texttt{i}) \Rightarrow \phi_{\texttt{Exc}_\texttt{i}}[\texttt{c} \leftarrow 0][\texttt{st(0)} \leftarrow \texttt{st(c)}])) \end{array}$$

$\psi^{pre}(\texttt{m})$ – *the specified precondition of method* `m`

$\psi^{post}(\texttt{m})$ – *the specified postcondition of method* `m`

$\psi^{exc}_{\texttt{m}}$ – *the exceptional function for method* `m`

$\texttt{numArgs(m)}$ – *the number of arguments of* `m`

$e_j, j = 1..s$ – *the locations modified by method* `m`

$\texttt{Exc}_\texttt{i}, \texttt{i} = 1..\texttt{k}$ – *the exceptions that* `m` *may throw*

$\phi_{\texttt{Exc}_\texttt{i}}, i = 1..k$ – *is the precondition of the exception handler protecting the instruction against* $\texttt{Exc}_\texttt{i}$ *if it exists, otherwiseif the* $\texttt{Exc}_\texttt{i}$ *is not handled*
$\phi = \psi^{exc}(\texttt{Exc}_\texttt{i})[\texttt{EXC} \leftarrow \texttt{st(c)}]$

Figure 7: *wp* rule for a call to an instance non void method

invocation on a non-void instance method is given in figure 7. In the precondition of the called method, the formal parameters and the object on which the method is called are substituted with the first *n+1* elements from the top of stack. Because the method returns a value, if it terminates normally, any occurence of the JML keyword `\result` in $\psi^{post}(m)$ is substituted with the fresh variable *fresh_var*. Because the return value in the normal case execution is put on the stack top, the *fresh_var* is substituted for the stack top in ψ. The resulting predicate is quantified over the expressions that may be modified by the called method. We also assume that if the invoked method terminates abnormally, by throwing an exception of type `Exc`, on returning the control to the invoker its exceptional postcondition $\psi^{exc}_{\texttt{m}}(\texttt{Exc})$ holds. The rule for static methods is rather the same except for the number of stack elements taken from the stack.

6.2 Abstracting The Control Flow Graph

In this section we discuss how the control flow graph of a bytecode is transformed in an acyclic control flow graph. We assume that the bytecode is provided with sufficient specification and in particular loop invariants. Under this assumption, we can "cut" the control flow graph at every program point where an invariant must hold and "place" at that point the invariant. This requires the introduction of several definitions.

A method body is an array of bytecode instructions. We write i_k the $k-th$ instruction in a method body. We assume that method's bytecode has exactly one entry point(every execution of a method starts at the entry point instruction) and we denote it with $i_{\texttt{entry}}$. Using standard terminology (see [2]), a basic block is a code segment that has no unconditional jump or conditional branch statements except for possibly the last statement, and none of its statements, except possibly the first, is a target of any jump or branch statement. We denote a block starting at instruction i_j with $\texttt{b}^\texttt{j}$. The block starting at the entry instruction is denoted with $\texttt{b}^{\texttt{entry}}$. The execution relation $\texttt{b}^\texttt{j} \rightarrow \texttt{b}^\texttt{k}$ states that block $\texttt{b}^\texttt{k}$ may be executed immediately after $\texttt{b}^\texttt{j}$ in some execution path of the method. For example if instruction $\texttt{i}_\texttt{k} = \texttt{athrow}$ is the last of the block $\texttt{b}^\texttt{j}$ then $\texttt{b}^\texttt{j} \rightarrow \texttt{b}^\texttt{n}$, where $\texttt{b}^\texttt{n}$ is the first block of an exception handler that protects $\texttt{i}_\texttt{k}$

Definition 1. Loop. Assume we have a bytecode Π. We say that $\texttt{b}^\texttt{e}$ is the entry block of a loop l in Π and $\texttt{b}^\texttt{f}$ is an end block of l and we note this with $\texttt{b}^\texttt{f} \rightarrow^l \texttt{b}^\texttt{e}$ if:

- every path in the control flow graph starting at the entry block $\texttt{b}^{\texttt{entry}}$ of Π and that reaches $\texttt{b}^\texttt{f}$, passes through $\texttt{b}^\texttt{e}$
- there is a path in which $\texttt{b}^\texttt{e}$ is executed immediately after the execution of $\texttt{b}^\texttt{f}$, $\texttt{b}^\texttt{f} \rightarrow \texttt{b}^\texttt{e}$

We abstract the execution relation $\rightarrow$ to the acyclic execution relation $\rightarrow^A$ which is an abstraction of $\rightarrow$ where the backedges $\rightarrow^l$ are removed: $\rightarrow^A = \rightarrow \setminus \rightarrow^l$

We give in Fig. 8 the control flow graph of the method `replace` given earlier in Fig. 2. The figure shows that the acyclic relation excludes edges between loop entry and loop end blocks and that at that place the corresponding loop invariant must hold.

6.3 Bytecode Weakest Precondition

In the following we overload the function symbol *wp* applying it to a method and sequence of bytecode instructions. The *wp* function runs in a backwards direction starting from the blocks that do not have successors up to reaching the entry point instruction. We distinguish the *wp* calculus over a method's body bytecode, a bytecode block and a sequence of bytecode instructions.

The weakest precondition $wp(\mathtt{m})$ *for method* $\mathtt{m}$ is the weakest precondition of its entry block $\mathtt{b}^{\mathtt{entry}}$.

$$wp(\mathtt{m}) = wp(\mathtt{b}^{\mathtt{entry}})$$

The weakest precondition for a bytecode block is calculated splitting the block in two parts: its last instruction and its sequential part (the instruction sequence without the last one). We note with $\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}$ the sequential part of block $\mathtt{b}^{\mathtt{i}}$ and $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}})$ the predicate that must hold after the execution of $\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}$ and before its last instruction; the weakest precondition of $\mathtt{b}^{\mathtt{i}}$ is then defined as follows:

$$wp(\mathtt{b}^{\mathtt{i}}) = wp(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}, post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}), \psi^{exc})$$

Concerning the sequential part, the *wp* is calculated applying the standard *wp* rule for compositional statements :

$$\begin{array}{l} wp(instrList; i_j, \psi, \psi^{exc}) = \\ wp(instrList, wp(i_j, \psi, \psi^{exc}), \psi^{exc}) \end{array}$$

The postcondition $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}})$ of the block $\mathtt{b}^{\mathtt{i}}$ depends on its last instruction and the respective predicates that must hold between $\mathtt{b}^{\mathtt{i}}$ and its successor blocks. Those predicates are determined by the function *pre*, which for any two blocks $\mathtt{b}^{\mathtt{i}}$ and $\mathtt{b}^{\mathtt{n}}$, such that $\mathtt{b}^{\mathtt{i}} \rightarrow \mathtt{b}^{\mathtt{n}}$ gives the predicate $pre(b^i, b^n)$ that must hold after the execution of $\mathtt{b}^{\mathtt{i}}$ and before the execution of $\mathtt{b}^{\mathtt{n}}$.

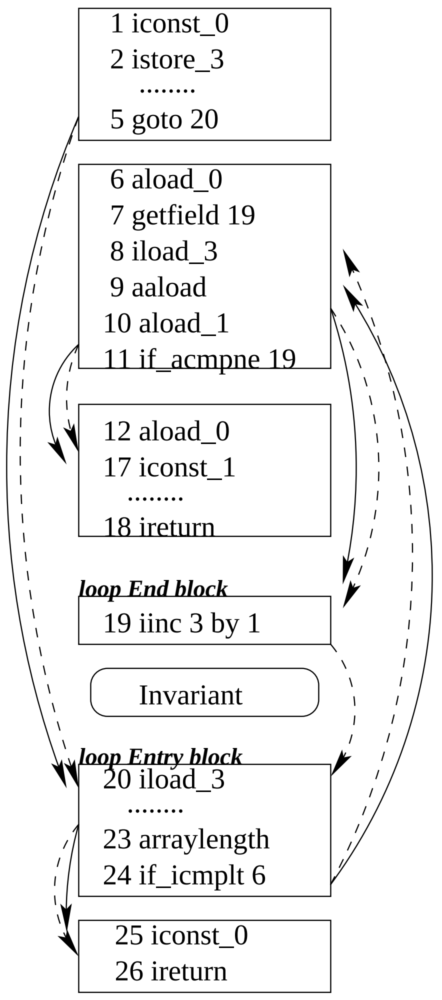
We define the postcondition of the sequential part of a block $\mathtt{b}^{\mathtt{i}}$ as follows:

Definition 2. Block's postcondition $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}})$. Let i_s be the last instruction of $\mathtt{b}^{\mathtt{i}}$ then:

- if $i_s = \mathtt{if\_cond\ n}$

$$post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}) = \begin{cases} cond(\mathtt{st(c)}, \mathtt{st(c-\ 1)}) \Rightarrow \\ \quad pre(b^i, b^n)[\mathtt{c} \leftarrow \mathtt{c} - 2] \\ \wedge \\ not(cond(\mathtt{st(c)}, \mathtt{st(c-\ 1)})) \Rightarrow \\ \quad pre(b^i, b^{s+1})[\mathtt{c} \leftarrow \mathtt{c} - 2] \end{cases}$$

- if $i_s = \mathtt{goto\ n}$
  $post(\mathtt{b}^{\mathtt{i}}_{\mathtt{seq}}) = pre(b^i, b^n)$
- if $i_s = \mathtt{athrow}$



dashed arrows stand for the standard execution relation
black arrows represent the acyclic execution relation
bytecode basic blocks are placed in boxes
the invariant is placed between the blocks where it must hold

Figure 8: control flow graph of method `replace` from figure 2

1. if there exists a block $\mathtt{b^e}$ such that $\mathtt{b^i} \to \mathtt{b^e}$ (i.e. an exception handler protects the type of the exception thrown) then :
   $post(\mathtt{b^i_{seq}}) =$
   $pre(b^i, b^e)[\mathtt{c} \leftarrow 0][\mathtt{st(0)} \leftarrow \mathtt{st(c)}]$.
2. Otherwise the thrown exception is not handled and then $\mathtt{b^i}$ must respect the postcondition determined by the exceptional postcondition function ψ^{exc} for this exceptional type:
   $post(\mathtt{b^i_{seq}}) =$
   $\psi^{exc}(\mathtt{st(c)})[\mathtt{c} \leftarrow 0][\mathtt{st(0)} \leftarrow \mathtt{st(c)}][\mathtt{EXC} \leftarrow \mathtt{st(c)}]$.
   see in section 5.1 for the meaning of `EXC`.

- if $i_s = \mathtt{return}$
  $$post(\mathtt{b^i_{seq}}) = \psi[\backslash\mathtt{result} \leftarrow \mathtt{st(c)}]$$
  where ψ is the specified method postcondition.
- else
  $post(\mathtt{b^i_{seq}}) =$
  $wp(i_s, pre(b^i, b^{s+1}), \psi^{exc})$

The function *pre* determines what is the property that should hold between two blocks that execute one after another, depending on if they determine a cycle or not (see definition 1 in the previous Section 6.2). This definition gives us the right to abstract the control flow to an acyclic one, as discussed in the previous subsection 6.2 and perform on the latter the weakest precondition calculus.

Definition 3. Predicate between consecutive blocks. Assume that $b^i \to b^n$. The predicate $pre(b^i, b^n)$ must hold after the execution of $\mathtt{b^i}$ and before the execution of $\mathtt{b^n}$ and is defined as follows:

- if $\mathtt{b^i} \to^1 \mathtt{b^n}$ then the corresponding loop invariant must hold:
  $$pre(b^i, b^n) = I$$
- else if $\mathtt{b^n}$ is a loop entry then the corresponding loop invariant I must hold before $\mathtt{b^n}$ is executed, i.e. after the execution of $\mathtt{b^i}$. We also require that I implies the weakest precondition of the loop entry instruction. The implication is quantified over the locations $m_i, i = 1..s$ that may be modified in the loop.
  $$pre(b^i, b^n) = I \wedge \forall_{i=1..s} m_i.(I \Rightarrow wp(b^n))$$
- else the normal precondition is taken into account:
  $$pre(b^i, b^n) = wp(b^n)$$

Subroutines are treated, first by identifying the instructions that belongs to it and then by in-lining them. Exception handlers are treated by identifying the instructions that belong to the handler (the class file format provides information about the exception handlers for all methods, in particular where starts and ends the region they protect and at what index the handler starts at); the precondition of the handler bytecode is calculated upon the normal postcondition of the method.

6.4 Verifying Java Bytecode Programs

Bytecode programs represent a set of Java classes. Establishing the correctness of Java bytecode program w.r.t. to their specification thus, consists in generating verification conditions for every method appearing in every class of the bytecode program. The verification procedure for a method `m` consists in calculating the weakest precondition $wp(\mathtt{m})$ upon its specification: precondition $\psi^{pre}(m)$, postcondition $\psi^{post}(\mathtt{m})$ and the mapping between exceptional types and predicates $\psi^{exc}(\mathtt{m})$ and then prove the condition:

$$\psi^{pre}(m) \Rightarrow wp(m)$$

The verification procedure does not trust neither the bytecode specification, nor the bytecode; in both cases — wrong specification or incorrect implementation will result in verification conditions that are not provable

We give a simple example of how the *wp* works. Block $\mathtt{b^6}$ (starts at instr. `6`) in Fig. 8 ends with a branching instruction and in the case when the condition is true (the current element of the array is not equal to the first parameter of the method `replace`) the execution will continue at $\mathtt{b^{19}}$. Below we give the part of the weakest precondition for block $\mathtt{b^6}$ in case the control flows to block $\mathtt{b^{19}}$ (the condition of its last instruction holds and in this case the predicate $pre(b^6, b^{19})$ is $wp(\mathtt{b^{19}})$). The implications with conclusion *false* stand for the possible exceptions `NullPointer` and `ArrayIndexOutOfBound` exceptions that may be thrown (as no postcondition is specified explicitely for these cases of abnormal termination, the one by default is taken).

$$\begin{array}{l}
\mathtt{lv[0]} = \mathtt{null} \Rightarrow \mathit{false} \wedge \\
(\mathtt{lv[0]} \neq \mathtt{null} \wedge \\
\#19(\mathtt{lv[0]}) \neq \mathtt{null} \wedge \\
\mathit{len}(\#19(\mathtt{lv[0]})) > \mathtt{lv[3]} \wedge \\
\mathtt{lv[3]} \geq 0 \wedge \\
\mathtt{lv[1]} \neq \#19(\mathtt{lv[0]})[\mathtt{lv[3]}]) \\
\Rightarrow \left(\begin{array}{l}
1 + \mathtt{lv[3]} \leq \mathit{len}(\#19(\mathtt{lv[0]})) \wedge \\
1 + \mathtt{lv[3]} \geq 0 \wedge \\
\forall v0. \left(\begin{array}{l}
0 \geq v0 \wedge \\
v0 < 1 + \mathtt{lv[3]} \\
\Rightarrow \\
\#19(\mathtt{lv[0]})[v0] \neq \mathtt{lv[1]}
\end{array} \right)
\end{array} \right) \wedge \\
\#19(\mathtt{lv[0]}) = \mathtt{null} \Rightarrow \mathit{false} \wedge \\
\mathit{len}(\#19(\mathtt{lv[0]})) \leq \mathtt{lv[3]} \vee \mathtt{lv[3]} < 0 \Rightarrow \mathit{false}
\end{array}$$

Here we discuss some experimental results. The JML compilation augments around twice the file size. For the example given at fig. 2, the class file without the specification extensions is 548 bytes, and the class with the BCSL extension BCSL is 954 bytes. Ofcourse, the more specific is the specification, the greater will be the size of the class file. On bytecode level, Jack generated 4 proof obligations, which can be proven interactively with Coq. On source level of the program 13 proof obligations, which can also be proven interactively in Coq. The proof obligations on source and bytecode level are basically the same modulo local variable names and field names; the difference in the number is due to differences in the implementation of *wp* on source and bytecode level; splitting the conjunctions in the bytecode proof obligations results in the same proof obligations generated over the source code (modulo names).

7. CONCLUSION AND FUTURE WORK

This article describes a bytecode weakest precondition calculus applied to a bytecode specification language (BCSL). BCSL is defined as suitable extensions of the Java class file format. Implementations for a proof obligation generator and a JML compiler to BCSL have been developped and are part of the Jack 1.8 release[3]. At this step, we have built a complete framework for Java program validation. This validation can be done at source or at bytecode level in a common environment: for instance, to prove lemmas ensuring bytecode correctness all the current and future provers plugged in Jack can be used.

We are now targeting to complete our architecture for establishing trust in untrusted code - in particular extending the present work to a PCC architecture for establishing non trivial requirements. In this way, several important directions for future work are:

- perform case studies and strengthen the tool with more experiments.
- find an efficient representation and validation of proofs in order to construct a PCC framework for Java bytecode. We would like to build a PCC framework where the proofs are done interactively over the source code and then compiled down to bytecode. Actually, observing the proof obligations generated over a source program and over its compilation with non optimizing compiler are rather similar (modulo names and certain types, e.g. boolean type).
- an extension of the framework applying previous research results in automated annotation generation for Java bytecode (see [19]). The client thus will have the possibility to verify a security policy by propagating properties in the loaded code and then by verifying that the code verify the propagated properties.

Finally, we are currently proving the correctness of the semantics of the weakest precondition calculus proposed, the proof is built over the bytecode operational semantics and will ensure the soudness of our weakest precondition calculus.

8. REFERENCES

[1] escjava. `http://secure.ucd.ie/products/opensource/ESCJava2/docs.html`.

[2] Aho AV, Sethi R, and Ullman JD. *Compilers-Principles, Techniques and Tools.* Addison-Wesley: Reading, 1986.

[3] Fabian Bannwart and Peter M:uller. A program logic for bytecode. 2005.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In Springer, editor, *in CASSIS workshop proceedings*, 2004.

[5] Nick Benton. A typed logic for stack and jumps. 2004.

[6] B.Meyer. *Object-Oriented Software Construction.* 2 rev edition edition, 1997.

[7] Richard Bornat. Proving pointer programs in Hoare Logic. In *MPC*, pages 102–126, 2000.

[8] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS 2003)*, volume 80 of *ENTCS*. Elsevier, 2003.

[9] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

[10] Lilian Burdy and Mariela Pavlova. From JML to BCSL. Technical report, INRIA, Sophia-Antipolis, 2004. draft.

[11] Lilian Burdy and Mariela Pavlova. Weakest precondition calculus for Java bytecode. Technical report, INRIA, Sophia-Antipolis, 2004. draft.

[12] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java modeling language.

[13] G.T.Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry. *JML Reference Manual.*

[14] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning.*

[15] Tim Lindholm and Frank Yellin. Java virtual machine specification. Technical report, Java Software, Sun Microsystems, Inc., 2004.

[16] G.C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119. ACM Press, 1997.

[17] George Necula. *Compiling With Proofs.* PhD thesis, Carnegie Mellon University, 1998.

[18] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI98*, 1998.

[19] M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In *CARDIS 2004*. Springer-Verlag, 2004.

[20] C.L. Quigley. A programming logic for Java bytecode programs. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[21] A.D. Raghavan and G.T. Leavens. Desugaring JML method specification. Report 00-03d, Iowa State University, Department of Computer Science, 2003.

[22] R.W.Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *volume 19 of Proceedings of Symposia in Applied Mathematics*, pages 19–32, 1967.

[3]http://www-sop.inria.fr/everest/soft/Jack/jack.html

[23] Martin Wildmoser and Tobias Nipkow. Asserting bytecode safety. 2005. to appear.